The Emperor's New Password Manager: Security Analysis of Web-based Password Managers

Zhiwei Li
Warren He
Devdatta Akhawe
Dawn Song

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2014-138
http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-138.html

July 7, 2014

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
07 JUL 2014		**00-00-2014 to 00-00-2014**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
The Emperor's New Password Manager: Security Analysis ofWeb-based Password Managers	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of California at Berkeley,Electrical Engineering and Computer Sciences,Berkeley,CA,94720**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

password managers. Unlike ?local? password managers web-based password managers run in the browser. We identify four key security concerns for web-based password managers and, for each, identify representative vulnerabilities through our case studies. Our attacks are severe in four out of the five password managers we studied an attacker can learn a user?s credentials for arbitrary websites. We find vulnerabilities in diverse features like one-time passwords, bookmarklets, and shared passwords. The root-causes of the vulnerabilities are also diverse ranging from logic and authorization mistakes to misunderstandings about the web security model, in addition to the typical vulnerabilities like CSRF and XSS. Our study suggests that it remains to be a challenge for the password managers to be secure; password managers in their current form, may not provide sufficient security for user secrets. To guide future development of password managers, we provide guidance for password managers. Given the diversity of vulnerabilities we identified we advocate a defense-in-depth approach to ensure security of password managers.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**20**	

The Emperor's New Password Manager: Security Analysis of Web-based Password Managers

Zhiwei Li, Warren He, Devdatta Akhawe, Dawn Song
University of California, Berkeley

Abstract

We conduct a security analysis of five popular web-based password managers. Unlike "local" password managers, web-based password managers run in the browser. We identify four key security concerns for web-based password managers and, for each, identify representative vulnerabilities through our case studies. Our attacks are severe: in four out of the five password managers we studied, an attacker can learn a user's credentials for arbitrary websites. We find vulnerabilities in diverse features like one-time passwords, bookmarklets, and shared passwords. The root-causes of the vulnerabilities are also diverse: ranging from logic and authorization mistakes to misunderstandings about the web security model, in addition to the typical vulnerabilities like CSRF and XSS. Our study suggests that it remains to be a challenge for the password managers to be secure; password managers, in their current form, may not provide sufficient security for user secrets. To guide future development of password managers, we provide guidance for password managers. Given the diversity of vulnerabilities we identified, we advocate a defense-in-depth approach to ensure security of password managers.

1 Introduction

It is a truth universally acknowledged, that password-based authentication on the web is insecure. One primary, if not *the* primary, concern with password authentication is the cognitive burden of choosing secure, random passwords across all the sites that rely on password authentication. A large body of evidence suggests users have—possibly, rationally [22]—given up, choosing simple passwords and reusing them across sites.

Password managers aim to provide a way out of this dire scenario. A secure password manager could automatically generate and fill-in passwords on websites, freeing users from the cognitive burden of remembering them. Additionally, since password managers automatically fill in passwords based on the current location of the page, they also provide some protection against phishing attacks. Add cloud-based synchronization across devices, and password managers promise tremendous security and usability benefits at minimal deployability costs [10].

Given these advantages, the popular media often extols the security advantages of modern password managers (e.g., CNET [11], PC Magazine [30], and New York Times [34]). Even technical publications, from books [12, 36] to papers [21], recommend password managers. A recent US-CERT publication [23] notes:

> [A Password Manager] is one of the best ways to keep track of each unique password or passphrase that you have created for your various online accounts without writing them down on a piece of paper and risking that others will see them.

Unsurprisingly, users are increasingly looking towards password managers for relieving password fatigue. LastPass, a web-based password manager that syncs across devices, claimed to have over a million users in January 2011 [27]. PasswordBox, launched in May 2013, claims to have over a million users in less than three months [44].

Our work aims to evaluate the security of popular password managers in *practice*. While idealized password managers provide a lot of advantages, implementation flaws can negate all the advantages of an idealized password manager, similar to previous results with other password replacement schemes such as SSOs [42, 40]. We aim to understand the current state of password managers and identify best practices and anti-patterns to guide the design of current and future password managers.

Widespread adoption of insecure password managers could make things worse: adding a new, untested single point of failure to the web authentication ecosystem. After all, a vulnerability in a password manager could

allow an attacker to steal *all* passwords for a user in a single swoop. Given the increasing popularity of password managers, the possibility of vulnerable password managers is disconcerting and motivates our work.

We conduct a comprehensive security analysis of five popular, modern web-based password managers. We identified four key concerns for modern web-based password managers: bookmarklet vulnerabilities, "classic" web vulnerabilities, logic vulnerabilities, and UI vulnerabilities. Using this framework for our analysis, we studied each password application and found multiple vulnerabilities of each of the four types.

Our attacks are severe: in four out of the five password managers we studied, an attacker can learn a user's credentials for arbitrary websites. We find vulnerabilities in diverse features like one-time passwords, bookmarklets, and shared passwords. The root-causes of the vulnerabilities are also diverse: ranging from logic and authorization mistakes to misunderstandings about the web security model, in addition to vulnerabilities like CSRF and XSS.

All the password manager applications we studied are proprietary and rely on code obfuscation/minification techniques. In the absence of standard, cross-platform mechanisms, the password managers we study implement features like auto-fill, client-side encryption, and one-time password in diverse ways. The password managers we study also lack a published security architecture. All these issues combine to make analysis difficult.

Our main contribution is systematically identifying the attack surface, security goals, and vulnerabilities in popular password managers. Modern web-based password managers are complex applications and our systematic approach enables a comprehensive security analysis (in contrast to typical manual approaches).

Millions of users trust these vulnerable password managers to securely store their secrets. Our study strikes a note of caution: while in theory password managers provide a number of advantages, it appears that real-world password managers are often insecure.

Finally, to guide future development of password managers, we provide guidance for password managers. We identify anti-patterns that could hide more vulnerabilities; architectural and protocol changes that would fix the vulnerabilities; as well as identify mitigations (such as Content Security Policy [14]) that could have mitigated some vulnerabilities. Our focus is not on finding fixes for the vulnerabilities we identified; instead, our guidance is broader and aims to reduce and mitigate any future vulnerabilities. Given the diversity of vulnerabilities we identified, we believe a defense-in-depth approach has the best shot at ensuring the security of password managers.

Ethics and Responsible Disclosure. We experimentally verified all our attacks in an ethical manner. We reported all the attacks discussed below to the software vendors affected in the last week of August 2013. Four out of the five vendors responded within a week of our report, while one (NeedMyPassword) still has not responded to our report. Aside from linkability vulnerabilities and those found in NeedMyPassword, all other bugs that we describe in the paper have been fixed by vendors within days after disclosure. None of the password managers had a bug bounty program.

Alice	a legitimate user
Bob	a legitimate collaborator
hunter2	an example password
`dropbox.com`	a benign web application
`facebook.com`	a benign web application
`/login`	entry point (login page) for a web application
Mallory	an attacker
Eve	an attacker
`evil.com`	a website controlled by an attacker
`dropbox.com`	The dropbox.com JavaScript code running in the browser

Figure 1: Naming convention used in the paper. URLs default to https unless otherwise specified.

Organization. We organize the rest of the paper as follows: Section 2 provides background on modern web-based password managers and their features. We also articulate their security goals and explain our threat model in Section 2. Next, we present the four key sources of vulnerabilities we used to guide our analysis (Section 3). Section 4 presents our study of five representative password managers, broken down by the source of vulnerabilities (per Section 3). We provide guidance to password managers in Section 5. We present related work in Section 6 before concluding (Section 7).

2 Background

To start, we explain the concept of a password manager and discuss some salient features in modern implementations. We also briefly list the password managers we studied, identify the threat model we work with, and the security goals for web-based password managers. Here and throughout this paper, we rely on a familiar naming convention (presented in Figure 1) to identify users, web applications, and attackers.

2.1 A Basic Password Manager

At its core, a password manager exists as a database to store a user's passwords and usernames on different sites. The password manager controls access to this database via a *master username/password*. A secure password manager, with a strong master password, ensures that a user can rely on distinct, unguessable passwords for each

web application without the associated cognitive burden of memorizing all them. Instead, the user only has to remember one strong master password.

A password manager maintains a database of a user's *credentials* on different *web applications*. A web application is a site that authenticates its users by asking for a username/password combination. The web application's "entry point" is the page where the application's user can enter her username and password. We call the combination of an entry point, username, and password a *credential*. A user can store multiple credentials for the same web application, in which case a name distinguishes each (typically the username).

Figure 2 (a) illustrates the general protocol of how a user (Alice) uses a password manager (e.g., LastPass) to log in to a web application (e.g., Dropbox). Alice first logs in to the password manager using her master username/password (her LastPass username and password), as shown in Step ①. Then, in Step ②, Alice retrieves her credential for `dropbox.com`. Finally, Alice uses this credential to log into `dropbox.com` in Step ③ and ④.

Since manually retrieving and sending credentials is cumbersome, password managers may also automate the process of selecting the appropriate credential and logging in to the opened web application. This may include navigating a web browser to the entry point, filling in some text boxes with the username/password, and submitting the login form. Since these tasks involve executing code inside the web application, password managers often rely on a privileged browser extension or a bookmarklet for the same.

2.2 Features in Modern Password Managers

Modern password managers provide a number of convenience and security features that are relevant to a security analysis. We briefly elucidate three below.

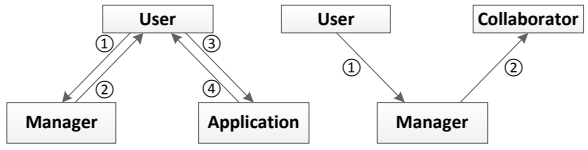

Figure 2: Different parties in a password manager scheme

Collaboration. Modern password managers include the ability to share passwords with a collaborator. Figure 2 (b) illustrates the general protocol of how a user Alice shares a credential of hers with a collaborator Bob. In Step ①, Alice requests that the password manager share a specified credential with Bob. In Step ②, the password manager forwards the credential to Bob when Bob requests it. Both Alice and Bob need accounts with the password manager. My1login even allows the password owner to set read/write permissions on the shared credentials, but the efficacy of these fine-grained controls is not clear, since denying write access does not prevent a collaborator from going to the web application and changing the account's password.

Credential Encryption. Due to the particularly sensitive nature of the data handled by password managers, password managers aim to minimize the amount of code and personnel with access to the credentials in the clear. One common technique is encrypting the credential database on the user's computer, thus preventing a passive attacker at the server-side from accessing the credentials in plaintext. In web-based password managers, this corresponds to using JavaScript to encrypt passwords on the client side (including pages on the password manager's website, browser extensions, and bookmarklets). The password manager encrypts/decrypts the credential database using a key derivation function starting from a user provided secret. If the password manager supports credential encryption, we call the encryption key the user's *master key*. For example, LastPass uses JavaScript to decrypt/encrypt the user's credential database using a key derived from the user's master username and password.

Login Bookmarklets. As discussed above, password managers typically rely on browser extensions to implement auto-fill and auto-login functionality. Unfortunately, users can only install these in a browser that supports extensions. With the popularity of mobile devices whose browsers lack support for extension APIs (e.g., Mobile Safari or Internet Explorer), password managers have adopted a more portable solution by providing a *bookmarklet*. A bookmarklet is a snippet of JavaScript code that installs as a bookmark, which, instead of navigating to a URL when activated, runs the JavaScript snippet in the (possibly malicious) context of the current page (e.g., `evil.com`). This allows the password manager to interact with a login form using widely supported bookmarking mechanisms.

2.3 Representative Password Manager Applications

To evaluate the security of modern password managers, we studied a representative sample of five modern password managers supporting a diverse mix of features. Table 1 provides an overview of their features. The columns "Extension" and "Bookmarklet" indicate support for login automation through the particular mechanism; "Website" indicates the presence of a web-based account management interface; and "Credential Encryption" and "Collaboration" refer to the features described

	Bookmarklet	Extension	Website	Credential Encryption		Collaboration
				Master Key Derivation	Encrypted Fields	
LastPass	✓	✓	✓	KDF(mp,mu,5000,32)	usernames and passwords	✓
RoboForm	✓	✓	✓	×		×
My1login	✓	×	✓	MD5(ph_{even})+MD5(ph_{odd})	usernames and passwords	✓
PasswordBox	×	✓	×	KDF(mp,mu,10000,32)	passwords only	✓
NeedMyPassword	×	×	✓	×		×

mu: master username mp: master password
ph: passphrase $ph_{even(odd)}$: characters at even (odd) positions of ph
KDF(p,s,c,l) is a key derivation function [25], which derives key of length l octets for the password p, the salt s, and the iteration count c.

Table 1: List of Password Managers Studied.

in Section 2.2. For password managers supporting credential encryption, Table 1 also lists their key derivation function and the fields encrypted.

2.3.1 LastPass

LastPass [26] is a popular, award-winning password manager available on phones, tablets, and desktops for all the major operating systems and browsers. It is the top-rated and Editors' Choice password manager for both PC Magazine [30] and CNET [11]. As of August 2013, LastPass had over one million users.

LastPass is one of the most full-featured password manager applications available. It supports nearly all major browsers and mobile/desktop platforms and includes features such as bookmarklets, one-time passwords, and two-factor authentication. LastPass users can access their credentials using the LastPass extension, through a bookmarklet, or directly through the LastPass website. LastPass stores the credential database encrypted on the LastPass servers and also allows users to share passwords with each other.

2.3.2 RoboForm

RoboForm (Everywhere) [35] is another top-rated password manager [30].[1] In RoboForm, each credential (i.e., username, password, and entry point tuple) has its own file named (by default) after the web application's domain. For example, RoboForm uses "dropbox" as the default filename when saving credentials for `dropbox.com`. The user can also choose arbitrary names for the files. Unless the user creates a master password to protect the files, these credential files are sent to RoboForm servers in the clear. The user can access her credential files directly through the RoboForm website or via the RoboForm extension or bookmarklet.

[1]RoboForm (Desktop) is a version of RoboForm that only stores credentials on a single computer and does not sync across devices using a web server. We focus only on the web-based RoboForm (Everywhere) software.

2.3.3 My1login

My1login is a web-based password manager, launched in April 2012; it started a special business-targeted product launched in May 2013. Our study was based on a then-beta version of their consumer-facing service. For maximum compatibility, My1login relies exclusively on bookmarklets and does not provide any browser extensions. Users can access credentials via a web application. My1login also supports sharing of credentials between two My1login accounts. My1login stores all credentials encrypted at the server-side with a special passphrase that the user sets up. In contrast to other password managers, which use the standard PBKDF algorithm, My1login concatenates the MD5 hash of odd and even characters of the passphrase to generate a 256-bit key. We do not comment on this further because we found a simpler, more severe flaw in My1login (Section 4.1.3) .

2.3.4 PasswordBox

PasswordBox [33], a web-based password manager that launched in 2013, is highly rated by both PC Magazine [30] and CNET [11]. Within three months of its inception in May 2013, PasswordBox had attracted over one million users [44]. PasswordBox, unlike other password managers discussed earlier, does not support bookmarklets; instead, it requires users to install a browser extension. PasswordBox also allows sharing credentials between users and encrypts all passwords using a 256-bit key derived using 10000 iterations of PBKDF2 and the PasswordBox username as the salt.

2.3.5 NeedMyPassword

Finally, we also studied a basic password manager named NeedMyPassword [32]. NeedMyPassword lacks common features such as auto-login, credential sharing,

and password generation. Instead, it provides only credential storage, accessible through the NeedMyPassword website. User credentials are not encrypted before sending to NeedMyPassword servers.

2.4 Threat Model

Our main threat model is the *web attacker* [2]. Briefly, a web attacker controls one or more web servers and DNS domains and can get a victim to visit domains controlled by the attacker. We believe this is the key threat model for web-based password managers that often run in the browser. For our study, we extend this model a bit: the user may create an account on the attacker's web application and use the password manager for managing the credentials for the same. Our threat model allows the victim to rely on the password manager's extension, the bookmarklet, and website as she sees fit. The attacker can also create accounts in the password manager service and make requests to the password manager directly.

The password manager's code often runs in a web application's origin (via an extension or a bookmarklet). We assume that the password manager's code is not malicious and does not steal sensitive data from web applications. We also assume that the password manager does not share Alice's credentials with user Bob, unless asked to do so by Alice. Additionally, we assume that the user uses a unique password for the password manager and does not share it with other applications such as `evil.com`.

2.5 Security Goal

At a high level, a password manager only has one key security invariant: ensure that a stored password is accessed only by the authorized user(s) and the website the password is for. We discuss how password managers (attempt to) achieve this invariant by following four security goals. A related taxonomy appears in Bonneau et al.'s analysis of general web authentication schemes [10], but ours is a bit different since we focus exclusively on web-based password managers. Nonetheless, all our goals map to goals mentioned in Bonneau et al.'s work. As we present in Section 4, we found attacks that violate *all* of the security goals identified below and range from complete (password manager) account takeover to privacy violations.

Master Account Security. The first goal of password manager application is the integrity of the master account. It should be impossible for an attacker to authenticate as the user to the password manager. It is crucial that the password manager maintain the security of the master account and safeguard credentials such as master password and cookies. In case of password managers that encrypt credentials, the master key/password used to encrypt the credential database should always remain at the client-side.

Credential Database Security. The main responsibility of a password manager is securely storing the list of a user's credentials. A password manager needs to ensure the security—including confidentiality, integrity, and availability—of the credential database. The attacker, Eve, should not be able to learn Alice's credentials, which would allow Eve to log in as Alice; or modify credentials, which would allow Eve to carry out a form of login CSRF attacks; or delete credentials, which would allow Eve to carry out a denial-of-service attack on Alice.

Collaborator Integrity. The collaboration, or sharing, feature in modern password managers complicates credential databases. Now, each credential has an access-control list identifying the list of users allowed to read-/write the credential. A password manager must ensure the security of this feature: e.g., flaws in this feature could allow an attacker to learn a user's credential. While we realize that these goals are a subset of the broader goal of credential database security (above), we separated them out to highlight the security concerns of the sharing credentials feature.

Unlinkability. The use of a password manager should not allow colluding web applications to track a single user across websites, possibly due to leaked identifiers. We use the Bonneau et al.'s definition of unlinkability [10]: a password manager violates unlinkability if it allows tracking a user across web applications even in the absence of other techniques like web fingerprinting [16]. For example, a privacy-minded user could rely on different browsers or computers to foil web browser fingerprinting; a password manager should not add a reliable fingerprinting mechanism that makes that effort moot. Such a fingerprinting mechanism would violate the user's privacy expectations. Equivalently, relying on a password manager should not allow a web application to link two accounts owned by the user with the (same) web application.

3 Attack Surface

The key difference between web-based password managers and "local" password managers is their need to work in web browsers. Web-based password managers store credentials in the cloud and a user logs on to the manager to retrieve his/her credentials. Access to the stored credentials is via extensions, a website, or even bookmarklets—all of which run in the browser.

To guide our investigation, we identified four key concerns for modern web-based password managers: bookmarklet vulnerabilities, classic web vulnerabilities, authorization vulnerabilities, and UI vulnerabilities. We discuss each in turn below. In the next section, we will present representative vulnerabilities of each type.

3.1 Bookmarklet Vulnerabilities

JavaScript is a dynamic, extensible language with deep support for meta-programming. The bookmarklet code, running in the context of the attacker's JavaScript context cannot trust *any* of the APIs available to typical web applications—an attacker could have replaced them with malicious code. Relying too much on these APIs has created a class of vulnerabilities unique to web-based password managers.

To fill in a password on (say) `dropbox.com`, a password manager needs to successfully authenticate a user, download the (possibly encrypted) credential, decrypt it (if necessary), authenticate the web application, and, finally, perform the login. Doing all this in an untrusted website's scripting environment (as a bookmarklet does) is tricky. In fact, three of the five password managers we studied (Table 1) provide full-fledged bookmarklet support, and *all* of them were vulnerable to attacks ranging from credential theft to linkability attacks (Section 4).

Browser extensions, which modified the webpage, faced a similar problem in the past. Currently, both Firefox and Chrome instead provide native or isolated APIs for browser extensions. Unfortunately, popular mobile browsers, including Safari on iOS, Chrome on Android/iPhone, and the stock Android Browser, do not support extensions. As a result, web-based password managers often rely on bookmarklets instead.

3.2 Web Vulnerabilities

A password manager runs in a web browser, where it must coexist with the web applications whose passwords it manages as well as other untrusted sites. Unfortunately, relying on the web platform for a security-sensitive application such as password managers is fraught with challenges.

Web-based password manager developers need to understand the security model of the web. For example, browsers share authentication tokens such as cookies across applications (including across applications and extensions), leading to attacks such as cross-site request forgery (CSRF). Applications running in the browser runtime also need to sanitize all untrusted input before inserting it into the document; insufficient sanitization could lead to cross-site scripting attacks, which in the web security model implies a complete compromise.

3.3 Authorization Vulnerabilities

Sharing credentials increases the complexity of securing password managers. While previously, each credential was only accessible by its owner, now each credential needs an access control list. Any user could potentially access a credential belonging to Alice, if Alice has authorized it. A password manager needs to ensure that all actions related to sharing/updating credentials are fully authorized. Confusing authentication for authorization is a classic security vulnerability, one that we find even password managers make (Section 4). We separate out authorization vulnerabilities from web vulnerabilities since they are often due to a missing check at the server-side. For example, all our authorization vulnerabilities involve requests made by an attacker from his own browser, not via Alice's browser (when Alice visits `evil.com`).

3.4 User Interface Vulnerabilities

A major benefit of password managers is their ability to mitigate phishing attacks. Users do not actually memorize the password for a web application; instead, they rely on the password manager to detect which application is open and fill in the right password. The logic that performs this is impervious to phishing attacks: it will only look at the URL to determine which credential to use.

These advantages are moot if the password manager itself is vulnerable to phishing attacks. Even worse, in the case of password managers, a single phishing attack can expose *all* of a user's credentials. Thus, we believe it behooves password managers to take extra precautions against phishing attacks. While it is possible that password managers are susceptible to classic phishing attacks, we focus on anti-patterns that make password managers more vulnerable than the typical website.

For example, consider what happens when a user clicks on a password manager's bookmarklet while not logged in to the password manager. A simple option is asking the user to login in an iframe. Unfortunately, this is trivial for the attacker to intercept and replace the iframe with a fake dialog. Since users cannot see the URL of an iframe, there is no way for a user to identify whether a particular iframe actually belongs to the password manager and is not spoofed. We argue that this is an anti-pattern that password managers should avoid.

4 Security Analysis of Web-based Password Managers

Next, we report the results of our security analysis of five popular password managers. We organize our results per the discussion in Section 3. Table 2 summarizes the vulnerabilities we found. Our discussion below highlights the presence of different types of security vulnerabilities in web-based password managers. We do not present complete architectural details of each password manager; instead, we only provide enough technical details to understand each vulnerability.

4.1 Bookmarklet Vulnerabilities

As discussed earlier, a bookmarklet allows a user of a password manager to log in to web applications without needing to install any extension, a particularly useful

	Bookmarklet Vulnerabilities	Web Vulnerabilities	Authorization Vulnerabilities	User Interface Vulnerabilities
LastPass	✓(§ 4.1.1)	✓(§ 4.2.1)		✓(§ 4.4)
RoboForm	✓(§ 4.1.2)	✓(§ 4.2.2)	NA	✓(§ 4.4)
My1login	✓(§ 4.1.3)		✓(§ 4.3.1)	
PasswordBox	NA		✓(§ 4.3.2)	NA
NeedMyPassword	NA	✓(§ 4.2.3)	NA	NA

Table 2: Summary of Vulnerabilities Discovered. NA identifies vulnerabilities not applicable to the particular password manager because it does not provide the relevant functionality.

feature with mobile browsers that lack extension support. Three of the password managers we studied—LastPass, RoboForm, and My1login—provide access to credentials and auto-fill functionality using bookmarklets. In fact, My1login *only* provides bookmarklet for auto-fill support, advertising it as a feature ("No install needed").

We found critical vulnerabilities in all three bookmarklets we studied. If a user clicks on the bookmarklet on an attacker's site, the attacker, in all three cases, learns credentials for arbitrary websites. .

While in 2009 Adida et al. identified attacks on password manager bookmarklets [1], our study indicates that these issues still plague password managers. This is particularly a cause of concern given the popularity of mobile devices that lack full-fledged support for extensions.

4.1.1 Case Study: LastPass Bookmarklet

LastPass stores the credential database on the `lastpass.com` servers encrypted with a `master_key`, which is a 256-bit symmetric key derived from the user's master username and master password. The LastPass client-side code never sends the master password or master key to the LastPass servers.

Recall that a bookmarklet runs in the context of the (possibly malicious) web application. At the same time, due to LastPass's credential encryption, the bookmarklet needs to include the secret `master_key` (or a way to get to it), to decrypt the credential database. Including this secret in the bookmarklet, while still keeping it secret from the web application, is tricky. LastPass also provides the ability to revoke a previously created bookmarklet, further complicating this feature.

Installing a Bookmarklet. A user, Alice, wishing to install a bookmarklet needs to create a special link through her LastPass settings page. On Alice's request, the LastPass page code creates a new random value `_LASTPASS_RAND` and encrypts the `master_key` with it, all within Alice's browser. The LastPass servers then store this encrypted master key (called `key_rand_encrypted`) and an identifier `h` along with Alice's credential database. The page then creates a JavaScript snippet containing `_LASTPASS_RAND` and `h`, which Alice can save as a bookmark. This design allows Alice to revoke this bookmarklet in the future by just deleting the corresponding `h` and encrypted master key from the LastPass servers.

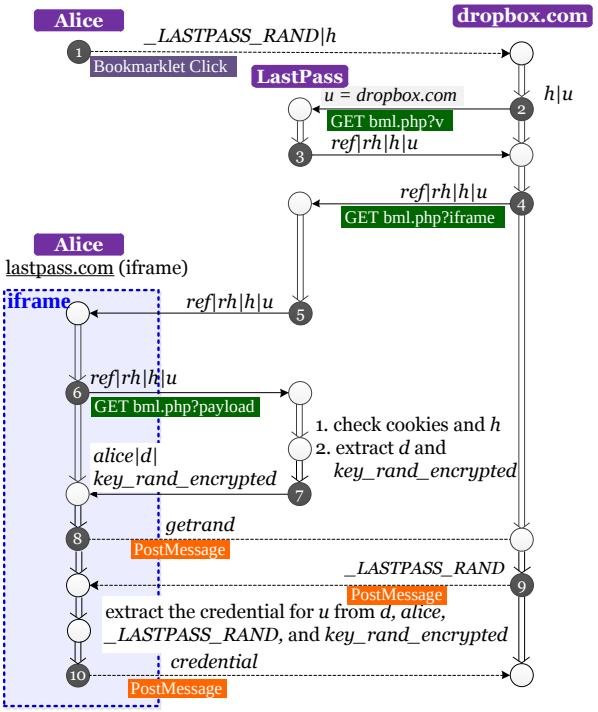

Figure 3: LastPass: Automatic login using bookmarklet. `u` is the domain on which Alice clicked on the bookmarklet.

Using the Bookmarklet. Figure 3 illustrates how Alice uses her LastPass bookmarklet to log in to `dropbox.com`. At the Dropbox entry point, Alice clicks on her LastPass bookmarklet, which includes the token `_LASTPASS_RAND` and `h`. The bookmarklet code first checks the current page's domain and adds a script element to the page sourced from `lastpass.com`. The request for the script element (Step 2 in Figure 3) sends `h` and the web application domain `dropbox.com` as parameters `h` and `u`. LastPass checks `h` and if the parameter is valid (i.e., Alice has not revoked the bookmarklet), re-

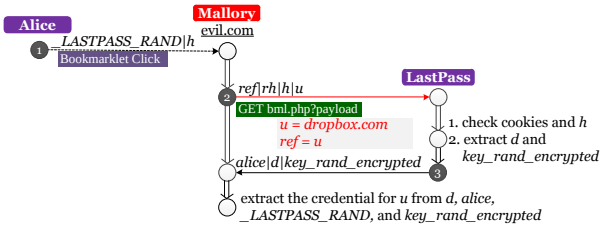

Figure 4: Attack on LastPass bookmarklet based autologin. The `rh,h` values are random; `u` and `ref` identify the Malloy's target website.

sponds with a JavaScript file containing the additional parameters `ref` and `rh`.

Next, the newly fetched JavaScript file creates an iframe to `lastpass.com` using four parameters: `ref,rh,h,u`. This iframe includes a script located at `lastpass.com/bml.php?u=dropbox.com` that, when downloaded, includes the encrypted master key `key_rand_encrypted` and the credential for `dropbox.com` encrypted with the master key. The iframe then receives the bookmarklet's `_LASTPASS_RAND` value via a `postMessage` call, decrypts the `dropbox.com` credential and sends them back.

Vulnerability. The resource at `bml.php?u=dropbox.com` (Step 6 Figure 3) is at a predictable URI and contains sensitive information. It provides the encrypted master key `key_rand_encrypted` and the credential for `dropbox.com`. The same-origin policy allows an attacker to include a script from any origin and execute it in the attacker's webpage.

LastPass Bookmarklet Attack. Figure 4 illustrates how a malicious web application `evil.com` can steal Alice's credential for `dropbox.com`. When Alice visits the attacker's site `evil.com` and clicks her LastPass bookmarklet, the attacker uses any of a number of hijack techniques [1, 8] (e.g., `Function.toSource`) and extracts both `h` and `_LASTPASS_RAND`. Then, the attacker imitates Step 6 from Figure 3 (as Step 2 here) by writing a `<script>` tag with `src` set to `lastpass.com/bml.php?u=dropbox.com` and adding the parameters `rh` (any string of length 64), `r` (any number), and `h` (from the bookmarklet).

The downloaded script, which runs on the attacker's page, includes all the information needed to decrypt credential for `dropbox.com` (notably, `key_rand_encrypted`). Again, the attacker uses the JavaScript hijack technique to extract out the encrypted credential and decrypts them with the `_LASTPASS_RAND` value stolen earlier. The attacker can repeat the attack to steal *all* of Alice's credentials, violating the confidentiality of the credential database.

LastPass Linkability Attack. Finally, we note that the `h` and `_LASTPASS_RAND` remain the same across browsers but differ by user. As discussed above, any website where the user clicks the bookmarklet can learn these pseudo-identifiers `h` and `_LASTPASS_RAND` [1]. This allows colluding websites to track a user, violating the user's privacy expectations [10]. Additionally, this also allows a single website to identify and link multiple accounts belonging to the same user, which violates the unlinkability goal.

4.1.2 Case Study: RoboForm Bookmarklet

RoboForm also implements a bookmarklet feature, which allows a RoboForm user to log in to web applications without installing the RoboForm extension. At a high level, the RoboForm bookmarklet adds an iframe to the web application login page. If Alice is already authenticated to RoboForm, the iframe just presents a list of valid credentials (RoboForm files) that Alice can use to log in to the web application.

```
1 javascript:(function(a,d,e){
  var c=function(){
3   return"function"===typeof a.srf7&&(a.srf7(e)||!0)
  },f=function(a){
5   ...
  };
7 c()||f(a.document)
})(top.self||window,"online.roboform.com/rfjs/main.js",
9 "d147b223873cbfca22bc66768db512b1")
```

Listing 1: RoboForm Bookmarklet code.

Technical Details. Figure 5 illustrates the RoboForm protocol if Alice, already logged in to RoboForm, clicks on the bookmarklet to log in to `dropbox.com`. When Alice clicks the RoboForm bookmarklet, the JavaScript snippet in Listing 1 executes in the current page. The hexadecimal number `d147...` (Line 9) is unique to Alice; we will refer to it as `rf_uid`. The bookmarklet code downloads a RoboForm script that creates an iframe to `roboform.com/rfjs/authform.php` (Step 2, Figure 5). The iframe URL also includes two GET parameters: `rf_referrer_url`, the URL of the web application's page, and `rf_uid`, Alice's pseudo-identifier.

The RoboForm server authenticates Alice with her cookies and, if Alice has valid session cookies, the RoboForm server redirects the iframe to `passcards.php` along with both `rf_referer_url` and `rf_uid`. `passcards.php` loads a list of filenames associated (Step 5, Figure 5) with the `rf_referer_url` (in this case, `dropbox.com`). Alice chooses a credential (i.e., a filename) and clicks "Fill Forms," sending the request in Step 6. Finally, RoboForm responds with Alice's credential (in the clear) in Step 7, and the iframe hands it to `dropbox.com` via `postMessage` in Step 8.

Vulnerability. In Step 8, the RoboForm iframe uses

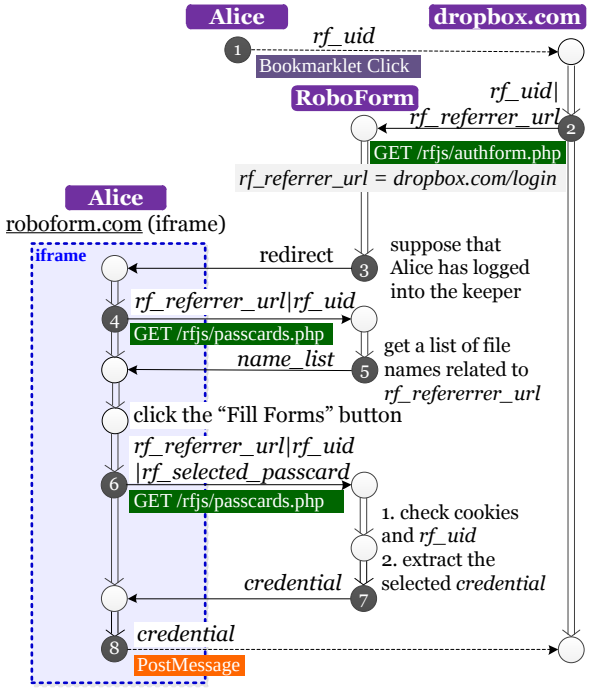

Figure 5: RoboForm Bookmarklet login when Alice is already logged in.

`postMessage` to send the credential without any origin check (i.e., a `*` target origin). Additionally, the attacker can modify the `rf_referer_url` parameter—used to identify the web application Alice wants to log in to—since the code runs in the context of the attacker's page.

RoboForm Bookmarklet Attack. Once Alice clicks on the bookmarklet on an attacker crafted webpage, the attacker can steal Alice's `rf_uid` and create an iframe (Step 2, Figure 5) with this `rf_uid` but `rf_referer_url` set to the target website (say, `dropbox.com`). The protocol proceeds as shown in Figure 5, except that the web application is actually `evil.com`. The message in Step 8 ends up sending credentials for `dropbox.com` to `evil.com`.

One complication arises between Step 5 and Step 6 in Figure 5. In Step 5, RoboForm responds with a list of possible credential files for the site identified by `rf_referer_url`. The filename defaults to the domain name the credential belongs to. This means that, for our attack to succeed, Alice needs to click on a file named `dropbox` while trying to log in to `evil.com`. (Un)Fortunately, we found a CSRF flaw in RoboForm that allows the attacker to rename the `dropbox` credential file to `evil`. To prevent later detection, the attacker can change the filename back to `dropbox` after the attack. We present full details of the CSRF flaw in Section 4.2.2.

RoboForm Linkability Attack. The bookmarklet exposes the value of `rf_uid` to `dropbox.com` as part of the iframe's `src` attribute. Colluding web applications can compare `rf_uid` values to link all of Alice's different credentials to a single user.

4.1.3 Case Study: My1login Bookmarklet

Unlike other password managers that use bookmarklets, My1login requires two steps for Alice to log in to `dropbox.com`. First, Alice visits her My1login master account page on `my1login.com` and clicks the link corresponding to Dropbox. This opens `dropbox.com` in a new window. Next, Alice clicks on her My1login bookmarklet. The JavaScript code in the bookmarklet then performs the automatic login. We informally call this two-step authentication process "click-then-fill."

Using a My1login bookmarklet. Figure 6 illustrates the protocol flow of My1login's "click-then-fill" authentication. When Alice clicks the link to Dropbox on `my1login.com`'s page, the page sends a POST request to My1login's server (Step 1) containing the credential id (called `webcardid` by My1login) for `dropbox.com`, the credential's encrypted username `alice` and password `hunter2`, and the decryption key `tmp_code`. The My1login server will hold the data for Alice to read in the next step. The My1login page also opens a new window to the entry point URL stored as part of the credential.

Next, when Alice clicks the My1login bookmarklet on the entry point window (Step 3), the bookmarklet loads JavaScript code from the URL `my1login.com/bookmarklet/1click_bm.js.php?url=dropbox.com`. This JavaScript file contains the credentials and decryption key saved in Step 1. The bookmarklet extracts the credentials, decrypts them using `tmp_code`, and fills in the login page with the username/password for `dropbox.com`.

Vulnerability. The salient vulnerability in Figure 6 is sending the decryption key (Step 1) to the My1login servers. Uploading the encrypted username and password to the My1login servers along with the decryption keys defeats the whole purpose of client-side encryption. Additionally, we found that the My1login server does not check that the `url` value in Step 4 corresponds to the web application the user clicked in Step 1 of Figure 6. In fact, the My1login server just ignores the `url` parameter and sends the username/password corresponding to the web card (i.e., credential) that Alice recently clicked on.

"Click-then-fill" Attack. Figure 7 illustrates a simple attack. Since the My1login server disregards `url` in Step 4, the attacker only needs to make a request, from `evil.com`, to `my1login.com/bookmarklet/1click_bm.js.php` after Alice clicked on Dropbox on her My1login page. The returned script provides Dropbox credentials to the attacker's page at `evil.com`.

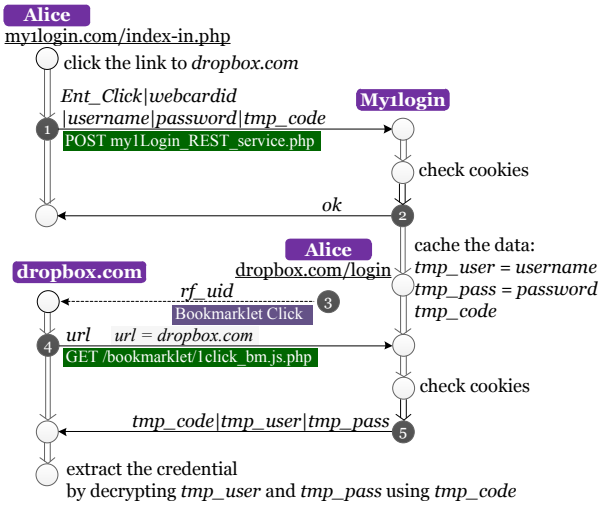

Figure 6: My1login: "Click-then-fill" authentication flow.

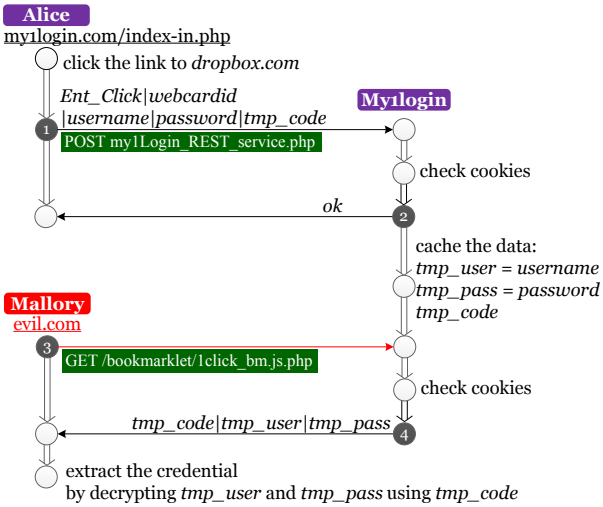

Figure 7: My1login: "Click-then-fill" attack

Two issues complicate the attack. First, since the request needs to happen from Alice's browser, Mallory (the attacker) must convince Alice to keep `evil.com` open in a background tab while logging in to `dropbox.com`. Second, the credentials stored at the My1login servers expire 90 seconds after the upload. This means that Step 3 in Figure 7 needs to happen no more than 90 seconds after Step 1.

We believe the prevalence of mashups on the web today means that this does not present a significant hurdle to Mallory. When Alice visit's Mallory's webpage at `evil.com`, it asks Alice to authorize the application via her Dropbox (or Twitter or Facebook) account. While Alice opens her My1login page in a new tab, the `evil.com` application (in the background) repeatedly requests `1click_bm.js.php`. This request successfully returns Alice's dropbox credentials as soon as Alice clicks on the Dropbox link on her My1login page.

Mallory's cross-origin request from `evil.com` to steal Alice's credentials causes the temporary credentials on the My1login server to expire. As a result, when Alice actually clicks on the bookmarklet in the newly opened Dropbox tab, the bookmarklet will ask Alice to open the My1login window and run the login protocol all over again. We believe that it is unlikely that this error will warn Alice about the attack.

Nevertheless, Mallory can also hide this indicator. Mallory can just "replay" the request in Step 1 with the just-stolen credentials. Mallory does not know the right `webcardid`, but we found that the My1login server-side does not check it. A random `webcardid` with a temporary username/password as well as the temporary decryption key that Mallory just stole is sufficient to "reset" the state at the server-side. After this request, Alice's login proceeds as normal without any indication to Alice about the credential theft. We stress that, while Mallory guesses the `webcardid`, she uses Alice's correct credentials for Dropbox (which Mallory just stole). Alice's login flow proceeds as normal, logging her in to her Dropbox account.

Linkability Attack. My1login is vulnerable to linkability attacks, like LastPass and RoboForm, due to its bookmarklet implementation. In Figure 6 Step 4, the My1login bookmarklet adds a `<script>` tag to the page with its src set to `my1login.com/bookmarklet/1click_bm.js.php?my1bmid=<somevalue>`, where the `my1bmid` value is a pseudo-identifier of the user. We did not present it in Figure 6 for clarity: the My1login servers do not seem to use the parameter value; instead, relying on cookies for identifying the user. Our "Click-then-fill" attack works without a `my1bmid` parameter.

4.2 Web Vulnerabilities

Next, we study vulnerabilities in password managers caused due to subtleties of the web platform. We focus on CSRF and XSS vulnerabilities, which are common in web applications. We find CSRF vulnerabilities in LastPass, RoboForm, and NeedMyPassword as well as XSS vulnerabilities in NeedMyPassword.

Our attacks are severe: XSS vulnerabilities in NeedMyPassword allow for complete account takeover, while the CSRF vulnerabilities in RoboForm allow an attacker to update, delete, and add arbitrary credentials to a user's credential database.

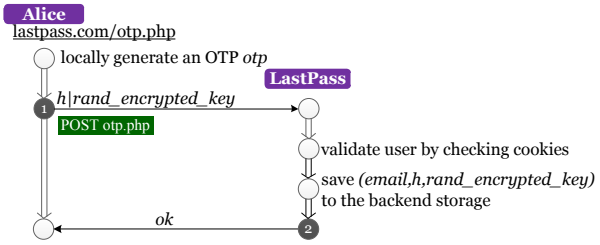

Figure 8: LastPass OTP Creation. Note the absence of any CSRF token in the request in Step 1.

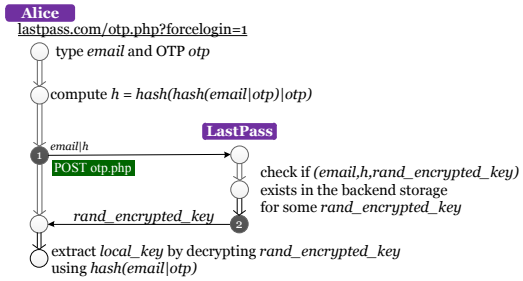

Figure 9: Using the LastPass OTP.`rand_encrypted_key` is the master key encrypted with `hash(alice|otp)`,

4.2.1 Case Study: LastPass One Time Password

One-Time password (OTP) is a feature of LastPass that allows a user to generate an authentication code for the master account that is only valid for one use. A user can use a one-time password to prevent a physical observer from gaining access to her LastPass account [10].

Generating an OTP. Before getting into the details, we point out that Alice's LastPass OTP must be able to authenticate Alice to LastPass *and* allow Alice to recover her master key; all without revealing anything extra (including the OTP itself) to LastPass servers (since that would defeat the credential encryption feature).

Figure 8 illustrates how Alice creates an OTP `otp`. This starts with Alice creating a string `otp` locally in her browser. Next, Alice computes `h = hash(hash(alice|otp)|otp)` with her LastPass username `alice`. LastPass will use `h` to authenticate Alice, without having to know the exact value of `otp`. Then, Alice encrypts her master key with `hash(alice|otp)`. Alice sends `h` and the encrypted master key (`rand_encrypted_key`) to LastPass. Notice that the LastPass servers never see the generated one-time password or Alice's master key in the clear. LastPass saves a record associating the values `h` and `rand_encrypted_key` with Alice's LastPass username.

Using the OTP. To sign in with her OTP (Figure 9), Alice recomputes `h` from her knowledge of `otp`, and sends it to LastPass along with her LastPass username. LastPass checks its records for a matching username and `h`. It starts an authenticated session for (i.e., sets session cookies identifying) Alice and sends back her `rand_encrypted_key`. Alice then decrypts `rand_encrypted_key` to recover her master key.

Vulnerability. We found that the request used to set up the OTP (Step 1 Figure 8) is vulnerable to a classic CSRF attack. The LastPass server authenticates Alice (in Step 1) only with her cookies. Since LastPass does not know the OTP or the master key, it cannot validate that `rand_encrypted_key` actually corresponds to an encrypted value of the master key. Fixing this vulnerability involves adding a CSRF token to the OTP creation form.

OTP Attack on LastPass. An attacker, Mallory, who knows Alice's LastPass username, can come up with a string `otp'` and using the same algorithm as above, generate a forged value `h'` and `rand_fake_key` with a made-up master key. On submitting the CSRF POST request, LastPass will store `h'` as authenticating Alice.

Mallory can then use `otp'` to log-in to LastPass using `otp'`. Of course, decrypting the `rand_fake_key` will not give Mallory Alice's real master key. Nonetheless, using this CSRF attack, Mallory obtains Alice's encrypted password database. We find this leads to three attacks.

First, LastPass stores the list of web application entry points unencrypted, and Mallory can now read this list. This is a breach of privacy: starting with just Alice's LastPass username, Mallory now knows all the web applications Alice has accounts on.

Secondly, the encrypted password database is now available to Mallory for offline guessing. Recall that the LastPass uses a key derived from Alice's *master* password, which Alice has to memorize. Unlike the passwords randomly generated by LastPass, this master password is likely vulnerable to guessing. It is instructive to consider that, after a server breach, LastPass requires all its users to reset their passwords [43].

Finally, we also find that this attack leads to a denial of service attack. Mallory, logged in as Alice, can delete any credential in Alice's database, *despite* being unable to decrypt the database. Since the username is part of the credential, recovering all these credentials would be tedious, or in some cases impossible.

4.2.2 Case Study: RoboForm Extension

RoboForm users can rely on a browser extension for accessing RoboForm functionality integrated into their browser. For example, an extension can automatically detect the current page and offer to fill in passwords instead of the user clicking on a bookmarklet. An ex-

tension has the additional advantage of running isolated from the main page—via the isolated worlds [20] in Google Chrome or native wrappers [31] in Firefox. This precludes a number of JavaScript hijack attacks that bookmarklets have to worry about.

While the RoboForm extension executes in a separate world, it still needs to interact with the `roboform.com` web servers. It still needs to worry about the typical threats that plague web applications. We examined the server APIs used by the RoboForm extension.

```
POST /requests/saveRFFile.php
Host: online.roboform.com
Origin: chrome-extension://kidhjpmgjfbkmcfpfakmdddd...
passcard_name=dropbox
& rf_referrer_url =www.dropbox.com/
&passcard_fields=#alice.dropbox@gmail.com#hunter2...

GET /requests/fileRename.php?...
    &f=%2Fdropbox.rfp&t=%2Fdropboxrenamed.rfp
Host: online.roboform.com
Origin: chrome-extension://kidhjpmgjfbkmcfpfakmdddd...

GET /requests/fileDelete.php?...&f=%2Fdropbox.rfp
Host: online.roboform.com
Origin: chrome-extension://kidhjpmgjfbkmcfpfakmdddd...
```

Listing 2: RoboForm requests vulnerable to CSRF.

Credential Manipulation. RoboForm stores credentials in text files on the `roboform.com` server. The extension uses the APIs listed in Listing 2 to manipulate the text files on the server. When Alice modifies a credential file and clicks the save button, the RoboForm extension POSTs to `saveRFFile.php` to save/update a file for `dropbox.com`. `passcard_name` is a user-chosen filename, which defaults to "dropbox" for `dropbox.com`, `rf_referrer_url` is Dropbox's entry point and `passcard_fields` contains Alice's username and password.

Updating or deleting an existing file also involves similar requests. We found that RoboForm uses GET requests to update or delete credential files, contrary to the intended semantics of the HTTP GET method. `fileRename.php` renames a file, where `f` and `t` correspond to the "from" and "to" filenames, respectively. `fileDelete.php` deletes a file with filename `f`.

Vulnerability. None of the requests for deleting, updating, or adding credentials included any CSRF tokens to protect against CSRF attacks. An attacker can issue these requests from her webpage and update/delete existing credentials or add new credentials to Alice's database.

RoboForm CSRF Attack. Mallory can simply craft a webpage that makes the relevant requests to the RoboForm servers and convince Alice to visit the page. For example, if Alice opens a webpage with `<img src="https://online.roboform.com/requests/fileRename.php?f=%2Fdropbox.rfp&t=evil.rfp"/>`, RoboForm will rename her "dropbox" credential to "evil". Similarly, Mallory can use these CSRF vulnerabilities to delete Alice's credentials (violating the availability goals). Mallory can also replace Alice's credentials with those under Mallory's control, leading to a form of login-CSRF attack.

Alice does not even need to use the RoboForm extension: these API endpoints work as long as Alice stays logged on to `roboform.com`. One mitigation that RoboForm could apply is using the Origin header sent by the Chrome extension. Unfortunately, no other browser sends this header for all requests and as a result, a complete solution requires a nonce-based CSRF defense.

These CSRF attacks do not disclose any sensitive information, but they violate the credential security, allowing an attacker to modify or delete credentials in the credential database. Moreover, as we discussed earlier, manipulating filenames is useful to prevent a victim from detecting a bookmarklet-based exploit.

4.2.3 Case Study: NeedMyPassword Website

As discussed earlier (Section 2.3), users access NeedMyPassword through its website, which stores a list of credentials that the user can update or view. While this limited functionality also reduces NeedMyPassword's attack surface, NeedMyPassword still needs to protect against standard web application security vulnerabilities such as XSS and CSRF.

Reviewing the NeedMyPassword website, we found that the NeedMyPassword webpage for displaying passwords (`www.needmypassword.com/passwordPresenter.php`) is vulnerable to reflected XSS. An attacker can pass an arbitrary value for the password parameter (`?password=<script>`) and take over the user's account. We also noticed that NeedMyPassword does not employ the `HttpOnly` mitigation for authentication cookies. This makes the attacker's job even easier: a call to `document.cookie` gives access to the Alice's session cookies, which Mallory can use to log in as Alice.

We also found CSRF vulnerabilities in the NeedMyPassword website. Listing 3 lists the HTTP requests to create, modify, and delete credentials, all of which are vulnerable to CSRF attacks. A number of attacks, similar to the ones we presented for the RoboForm CSRF vulnerabilities, are possible. For example, a CSRF attack on the delete functionality allows an attacker to delete all of Alice's passwords, a form of denial-of-service attack.

4.3 Authorization Vulnerabilities

Looking beyond vulnerabilities stemming from the nature of the web platform, we now discuss some vulnera-

POST /mypass.php
Host: needmypassword.com

system=dropbox&username=alice.dropbox@gmail.com
&password=hunter2&enter_password=enter

POST /editpass.php?id=21856
Host: www.needmypassword.com

system=dropbox&username=alice.dropbox@gmail.com
&password=bobpwd&save=Save

POST /editpass.php?id=21856
Host: www.needmypassword.com

system=dropbox&username=alice.dropbox@gmail.com
&password=bobpwd&delete=Delete

Listing 3: CSRF Vulnerable NeedMyPassword requests to set update and delete credentials.

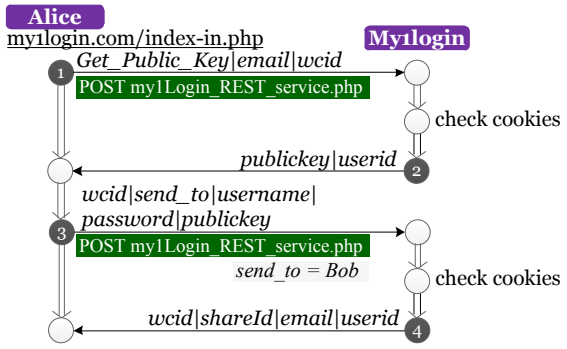

Figure 10: Sharing Credentials on My1login

bilities that come from logic errors in the password manager. We found that two of the three password managers that support credential sharing both mistake authentication for authorization. An attacker can create two fake accounts, Eve and Mallory, in the password manager and share Alice's credentials with Mallory by sending a correctly crafted message from Eve's account. Importantly, the actual errors do not ever involve Alice or her browser and thus the attacks work in the absence of Alice visiting the attacker's website.

4.3.1 Case Study: My1login Sharing Credentials

My1login relies on client-side encryption of the credential database. This complicates sharing: Alice and Bob need to share credentials, through My1login as an untrusted channel. My1login relies on public-keys for both Alice and Bob to share credentials: when Alice shares a credential with Bob, My1login first encrypts it with Bob's public-key before sending it to Bob. This ensures that only Bob can see the shared credentials.

Sharing My1login Credentials. Figure 10 illustrates how Alice shares a credential with Bob in My1login. In the first two steps, Alice obtains Bob's public key k_b. Then, in Step 3, Alice (i.e., Alice's My1login instance) encrypts the credential with k_b and sends the encrypted username `alice.dropbox@gmail.com` and password `hunter2` to My1login.

Using the Shared Credential. Bob's My1login instance polls the My1login server for any updates. The My1login server notifies Bob of the newly shared credential, sending him the information that Alice encrypted with his public key. Bob decrypts the shared credentials (`username` and `password`) for website `url` with his private key. Once Alice shares a credential with Bob, he can also update it. In such cases, My1login automatically updates the credential globally by sharing the update with collaborators on the web card (Alice, in this case). This occurs through essentially the same request as Step 3 in Figure 10, but this time Bob encrypts the credential with Alice's public-key.

Vulnerability. Our analysis revealed that My1login only *authenticates* Alice before sharing a web card; it does not check whether Alice owns or has the authority to share the web card identified in the `wcid` (Step 3, Figure 10).

My1login Share Attack. Since My1login does not check `wcid` in Figure 10 Step 3, an attacker Mallory can share any web card (given its id) to a collaborator Eve. This vulnerability allows Mallory to steal any credential whose ID she knows (perhaps because Eve shared it in the past but revoked it later).

Worse, further analysis revealed that web card ids are globally unique, auto-incrementing numbers. In Step 3, Figure 10, Mallory can even use numbers referring to cards not yet created.

Suppose that `wcid` refers to a web card that belongs to (or will belong to) Alice. Mallory generates a dummy username and password like "userabc" and "pwdabcm," encrypts it and shares it with Eve. Eve receives the dummy credentials. While these credentials are useless, notice that this registered Eve as a collaborator on this web card, even if it belongs to Alice.

In the future, whenever Alice or any other collaborator updates the web card, the My1login client automatically re-encrypts the real credential and sends it to each collaborator, *including Eve*. It is trivial for Mallory to share all web cards, current and future, to Eve, who awaits updates to steal real credentials.

In the attack above, Eve learns Alice's credentials only if Alice updates them after the attack. Alternatively, Eve can install new credentials to Alice's database without authorization from Alice. This allows Eve to execute a form of login CSRF attack [5]. Alternatively, Eve can install wrong credentials to Alice's database, which would cause an error when Alice attempts to use them. It is likely that Alice, in response, would update the web card with her correct credentials and unknowingly share them with Eve.

```
{ "id": 4097211,
  "member_id": 3751238,
  "name": "Dropbox",
  "url": "https :// www.dropbox.com/login",
  "login": "alice .dropbox@gmail.com",
  "note": {},
  "created_at": "2013-07-18T13:50:18-04:00",
  "updated_at": "2013-07-18T13:50:18-04:00",
  "password_k": "AAQsrfjgfcWj/4FsP64BTYTJpbgpBK4+yltal",
  "settings": "{\"autologin\":\"1\", ...}",
  "member_fullname": "Alice Gordon",
}
```

Listing 4: Example PasswordBox asset

One concern is how to ethically verify the My1login authorization flaw without sharing another user's credential by mistake. We observed over multiple days that it is rare that any other user creates a new web card between 2am - 3am PST. We then verified this vulnerability one day between 2am and 3am without sharing another user's credential by mistake.

4.3.2 Case Study: PasswordBox Sharing Credentials

PasswordBox stores a user's credential for a web application in a JSON-encoded *asset* file. Listing 4 presents an example asset for Dropbox. We focus on two salient properties: first, `password_k` is the encrypted value of Alice's password for `dropbox.com` and is the *only* encrypted field in the asset. Other details such as entry point URL, the name Alice used to register (`member_fullname`) and so on, are all in cleartext.

Second, our analysis revealed that `asset_id` is an auto-incrementing, unique (across all users) id for each asset. Assuming `asset_id` started at 1, we can infer that PasswordBox manages over 4 million assets, an assumption anyone can verify with the flaw we discuss next. (We did not, because of the obvious ethical concerns.)

Sharing Credentials. Figure 11 shows how a user Alice shares one of her assets identified by `asset_id` to a collaborator Bob. On clicking share, the PasswordBox extension on Alice's browser makes a POST request to the `passwordbox.com` servers that includes the `contact_id`, the contact to share credentials with (in this case, Bob's id); and `asset_id`, the id of the credential to share (as in Listing 4). In the future, whenever Bob downloads the list of assets accessible to him, PasswordBox includes Alice's shared credential.

Vulnerability. The absence of a CSRF token suggested the possibility of a CSRF flaw in the protocol. Fortunately (or, unfortunately), we found that PasswordBox implemented a strong defense against CSRF attacks: it checks the `Referer` header as well as includes a special `X-CSRF-Token` in the headers of the HTTP request. Instead, we found a far more serious logic bug in the sharing assets functionality. In its sharing

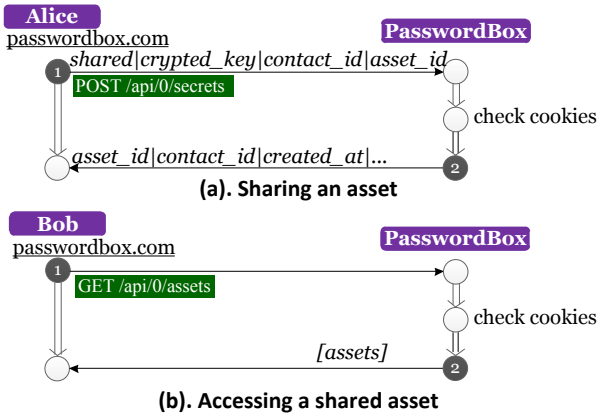

Figure 11: PasswordBox: Sharing an asset. The underlined `passwordbox.com` on the left indicates that the code making the request runs in the `passwordbox.com` origin.

```
function share(asset_id){
  var xmlhttp = new XMLHttpRequest();
  var jsn = '{"shared":true, "crypted_key:" "ABC", "contact_id": 123,
    "asset_id": '  + asset_id + '}';
  xmlhttp.open("POST","https://api0.passwordbox.com/api/0/secrets",true);
  xmlhttp.setRequestHeader("Content-type", "application/json");
  xmlhttp.send(jsn);
}
```

Listing 5: JavaScript snippet to share a asset with Eve

logic, PasswordBox never checks whether Alice owns the `asset_id` she is sharing. This allows Mallory to share assets she does not own with Eve, similar to the My1login attack (Section 4.3.1).

PasswordBox Share Attack. Similar to the "share-and-update" attack on My1login, Mallory and Eve run through the protocol in Figure 11. Mallory can share any asset to Eve by simply setting `asset_id`. Since `asset_id` is an auto increment number, Mallory can iterate through all possible `asset_id` and share all existing 4 million assets with Eve. Listing 5 is the JavaScript snippet that Mallory used to share an arbitrary asset to Eve, whose `contact_id` is assumed to be 123.

As we noted above, PasswordBox only encrypts the password field in an asset; disclosure of every user's full name, usernames, web application URLs, and creation times is a severe privacy breach.

4.4 User Interface Vulnerabilities

Earlier, discussing bookmarklet vulnerabilities (Section 4.1), we focused on the behavior of a password manager when the user is already authenticated to the password manager. If the user is not authenticated to the password manager, then the user needs to log in to her mas-

ter account. This provides a potential avenue for phishing vulnerabilities and the password manager should not train bookmarklet users towards insecure practices. The ideal secure option in such a scenario is asking the user open a new tab (manually) and logging in to the password manager.

We find that only the My1login bookmarklet defaults to this secure behavior. Clicking on the My1login bookmarklet, when not logged in, results in a message asking the user to open a new window and log in. We found that both RoboForm and LastPass bookmarklets were vulnerable to phishing attacks. We also have recorded video demonstrations of these attacks online [4].

Case Study: RoboForm. Recall that when Alice clicks her RoboForm bookmarklet, the bookmarklet creates an iframe in the current web application. If Alice has not logged in to RoboForm, the iframe request redirects to the RoboForm login page, displaying a login form in the iframe. This design is insecure: it trains Alice to fill in her RoboForm password even when the URL bar (belonging to the surrounding web application) does not point to `roboform.com`. An attacker can trivially block the RoboForm iframe load and spoof an authentication dialog that steals Alice's RoboForm credentials. A secure design would ask Alice to open a new tab to RoboForm and log in.

One concern with successfully carrying out this attack is detecting whether Alice is already logged in to RoboForm. We found that the height of the RoboForm iframe (the dialog) is greater than 200px if and only if Alice is already logged-in. Using this side-channel, the attacker can modify the spoofed iframe to make the attack convincing.

Case Study: LastPass. In the case of LastPass bookmarklet, if Alice is not already logged in, the bookmarklet code shows a message asking the user to log in, along with a link that opens the LastPass login page in a popup. While this design at least allows the browser to display the URL of the popup, quirks of the web platform make even this design less secure than using a manually opened window.

Recall that the attacker, Mallory, can subvert the LastPass bookmarklet and show an attacker controlled message to Alice. The key now is for Mallory to trick Alice into entering her LastPass credentials on an attacker-controlled page. We refer to an attack initially described by Zalewski [45]. Mallory creates a link to the normal LastPass login page, and when Alice clicks on it, a new window pops up at the LastPass login URL. Alice can check the URL bar and verify that the location is `https://lastpass.com`. Meanwhile, Mallory holds on to a handle to the window and can navigate the window to a page from Mallory's server. After an appropriate delay for Alice to check the URL bar, Mallory navigates the pop-up to a malicious page that looks identical to the LastPass login page.

Normally, Alice would notice a navigation to `evil.com`. Mallory can reduce the visibility of this navigation by navigating to a `data:` URI, which loads instantly. Mallory can also use modern features such as the `prerender` link directive to hide the navigation. The `data:` URI needs to look the same as the LastPass login page but with the login form submission target set to `evil.com`. We found that this attack is especially effective on Firefox for Android, since it only shows the page title and not the URL. Further, this is only an example attack. A number of other phishing attacks are practical on mobile device; Felt and Wagner provide a detailed discussion of the same [18].

5 Lessons and Mitigations

We now attempt to distill the lessons learnt from our study and provide guidance to password managers on closing the vulnerabilities we found and mitigating future ones. Our focus here is on concrete guidance and defense-in-depth. We identify improvements in architectures and protocols to mitigate vulnerabilities as well as the use of browser mitigations like CSP. We also identify anti-patterns that developers of password managers should avoid. Security reviewers and users can also rely on the patterns and (absence of) the mitigations we discuss as indicators of the security of a password manager.

5.1 Bookmarklet Vulnerabilities

All the bookmarklets we studied were vulnerable. The root cause of these vulnerabilities is that the bookmarklet code executes in the untrusted context of the webpage. The web browser guarantees a secure, isolated execution environment for iframes and we advocate an iframe-based architecture for securing password manager bookmarklets. Modern features such as credential encryption, which requires secure client-side code execution, makes the use of defenses proposed in previous work impractical [1].

Recommendation. We recommend password-managers rely on the design proposed by Bhargavan et al. [8]. When the user clicks the bookmarklet, the bookmarklet code loads the password manager code in an iframe, running in the password manager's origin. The browser's same-origin policy isolates code executing in the iframe from the web application page and guarantees integrity of DOM APIs.

The password manager's iframe uses `postMessage` for communicating with the application page and maintains a simple invariant: a message carrying a credential for dropbox.com has a target origin of `https://www.dropbox.com`. The browser guarantees that only the Dropbox page receives the message. The only se-

cret in the bookmarklet code is an HMAC function (protected by DJS [8]) that the password manager iframe can use to provide *click authentication* (i.e., the user actually clicked the bookmarklet). Unfortunately, the presence of the secret in the bookmarklet allows linkability attacks.

For unlinkability, we recommend password managers do not rely on such a secret and HMAC function. Disabling this secret loses the "click authentication" property. Since password manager browser extensions typically include "auto fill" functionality, we believe the loss of click authentication is acceptable. If needed, the code in the password manager iframe could draw a dialog to ask for user confirmation before sharing credentials with the website. Such a design is vulnerable to clickjacking and we also recommend the use of upcoming mitigations for UI security [41].

Instead, password managers could rely on asking the user for permission to share credentials in the iframe created.

The core issue behind bookmarklet vulnerabilities is the absence of secure (or "isolated") DOM APIs for bookmarklets. An alternative possibility is for browser vendors to provide bookmarklets with secure access to these DOM APIs, similar to the access granted to Chrome/Firefox extensions.

5.2 Web Vulnerabilities

We found a number of "classic" web application vulnerabilities in password managers. Based on the critical and sensitive nature of data handled by password managers, we recommend defense-in-depth features such as CSP and identify anti-patterns that developers should beware of.

XSS. XSS is a well-studied problem and we will not recapitulate all the defenses for the same here. We recommend that web applications, in addition to validating input and sanitizing outputs, should also turn on Content Security Policy to provide a second layer of defense against XSS. The absence of a strong CSP policy in a password manager should raise red flags for users and reviewers. In the applications we studied, only LastPass shipped with a Content-Security-Policy header, albeit with an unsafe policy that allows eval and inline scripts.

CSRF. The prevalence of CSRF vulnerabilities in password managers surprised us. We recommend password managers should include CSRF protection (via tokens) for *all* their pages and forms. For defense in depth, these applications should also check the Referer and Origin headers for all requests. While not a reliable defense, these headers provide a useful secondary layer of defense.

One concern with CSRF tokens is the need to create and maintain state at the server-side. This could be cumbersome for password managers that provide an interface through a browser extension: it is infeasible to request a new token before rendering every form. Instead, these applications can rely on special headers (e.g., X-CSRF-Token) for CSRF defense. The web security model disallows `evil.com` from setting headers for a cross-origin request.[2]

Secrets in JavaScript files. An anti-pattern we noticed was the presence of secret values—based off of tokens in the request URI or cookies in the request—in script files. Unfortunately, the web platform does not provide strong isolation guarantees for scripts: any (untrusted) origin can include scripts from the password manager's website. We recommend password managers serve *all* secret values in HTML or separate JSON files. This requirement is easy to check: the scripts used by the password managers should be the same across all users of the password manager. Serving user-specific JavaScript files based on tokens in the URI is a clear anti-pattern. An alternative is Defensive JavaScript [8], which provides a principled defense to ensure secrecy of values in JavaScript code.

5.3 Authorization Vulnerabilities

The web application vulnerabilities discussed above stemmed from quirks of the web platform (e.g., ambient authentication with cookies). Worryingly, we found a number of *logic* flaws in password managers classified under two broad categories. The first category, insufficient authorization, creates vulnerabilities exacerbated by the second category, predictable identifiers. We identify an anti-pattern, predictable identifiers, and the core security vulnerability, insufficient authorization, below and discuss mitigations.

Insufficient Authorization. Confusing authentication with authorization is a classic security vulnerability. Out of the three password managers that support collaboration, we found insufficient authorization vulnerabilities in two of them. Unfortunately, these are logic flaws, and a simple mitigation is difficult. One possibility is for password managers to use a simpler sharing model. For example, let each credential have only one owner—only the credential's owner can change it or its collaborator list. A simple model eases authorization checks and could make insufficient authorization stand out.

Predictable Identifier. Both our attacks on logic vulnerabilities rely on predictable identifiers (e.g., consecutive integers). We recommend password managers switch to cryptographically secure random numbers for identifiers—this adds defense in depth, even if the server is careful to check authorization. The use of predictable identifiers should be rare and any use should be a cause

[2]Unless explicitly whitelisted by the receiving server via Access-Control-* headers.

for a security review. As we discussed earlier, the nature of the data handled by password managers warrants such a default-secure posture.

5.4 User Interface Vulnerabilities

Our proposed solution of relying on iframes and storing tokens in localStorage/cookies works seamlessly only if the user is already logged in. If this is not true, the iframe needs to ask the user to log in. As our attacks demonstrated, the only secure way to do this is asking the user to manually open a new tab and login. My1login is the only password manager relying on this design and we recommend other password managers adopt a similar design. Cautious users can protect themselves against such an attack by always logging in using a new tab instead of trusting a popup or iframe.

6 Related Work

A number of researchers have investigated security of web-based password managers. Bhargavan et al. did a study on five password managers, along with a number of other web services that provide encrypted storage of data in the cloud, and presented a number of web attacks that could violate the intended security of the products [7]. This work inspired a redesign of the LastPass bookmarklet to decrypt a user's credentials inside LastPass's iframe, making it harder for an attacker to steal the master key. Adida et al. provide a comprehensive overview of a number of attacks on password manager bookmarklets; we reuse some of the ideas but find that, with modern password managers relying on encrypted credentials, a new defense based on iframes is needed [1]. Belenko et al. studied the cryptographic properties of password managers for mobile devices and their vulnerability to brute force attacks [6].

In concurrent work, Blanchou and Youn [9] as well as Silver et al. [37] found a number of serious flaws in the auto-fill functionality in password managers. In contrast, we analyze a broader range of functionality but focus on third-party web-based password managers only.

Bonneau et al. [10] presented a framework for evaluating alternatives to passwords in terms of usability, deployability, and security. This framework highlights advantages of an idealized password manager, but our work demonstrates that, in practice, password managers have flaws in their implementations that critically undermine their security. Similarly, recent work found implementation flaws in other password alternatives such as SSOs [42, 40].

The common web attack vectors we considered, such as CSRF and XSS, have seen a lot of work in the past decade. For attacks and defenses, we defer to prior literature for comprehensive surveys on CSRF [46], XSS [19], and server-side defenses for both [28]. Recent work also focused on logic flaws and insufficient authorization in web applications [17, 39, 38].

The security of mutually distrusting JavaScript running in the same origin (an important consideration in bookmarklet code) has not been a concern in the design of the web platform. Bhargavan et al. identified a number of flaws in bookmarklets and proposed a new subset of JavaScript called Defensive JavaScript to mitigate them, which we discussed in depth in Section 5.1. Defensive JavaScript [8] is the only work we are aware of that aims to protect a JavaScript gadget from the host webpage. A large body of work exists for the converse goal of protecting a host webpage from third party JavaScript code (such as code that draws a gadget) [24, 3, 13, 29]; a survey compares these approaches [15].

7 Conclusions

We presented a systematic security analysis of five web-based password managers. We found critical vulnerabilities in all the password managers and in four password managers, an attacker could steal arbitrary credentials from a user's account. Our work is a wake-up call for developers of web-based password managers. The wide spectrum of discovered vulnerabilities, however, makes a single solution unlikely. Instead, we believe developing a secure web-based password manager entails a systematic, defense-in-depth approach. To help such an effort, we provided guidance and mitigations based on our analysis. Since our analysis was manual, it is possible that other vulnerabilities lie undiscovered. Future work includes creating tools to automatically identify such vulnerabilities and developing a principled, secure-by-construction password manager.

Acknowledgements

We thank the anonymous reviews for their valuable feedback. We also thank Karthikeyan Bhargavan, David Wagner, Weichao Wang, Paul Youn, Chris Grier, Kurt Thomas, Matthew Finifter, Joel Weinberger, Chris Thompson, Suman Jana, and Nicholas Carlini for their valuable feedback and comments. This research was supported by Intel through the ISTC for Secure Computing; by the Air Force Office of Scientific Research (AFOSR) under MURI award FA9550-09-1-0539; by the Office of Naval Research (ONR) under MURI Grant N000140911081; and by the National Science Foundation (NSF) under grant 0831501CT-L. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF, the AFOSR, the ONR, or Intel.

References

[1] B. Adida, A. Barth, and C. Jackson. Rootkits for javascript environments. In *Proc. of WOOT 2009*, 2009.

[2] D. Akhawe, A. Barth, P. E. Lam, J. Mitchell, and D. Song. Towards a formal foundation of web security. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium*, 2010.

[3] D. Akhawe, P. Saxena, and D. Song. Privilege separation in html5 applications. In *Proc. the 21st USENIX Security symposium*, 2012.

[4] Ui attacks demos, 2013. `https://sites.google.com/site/webpwdmgr/`.

[5] A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In *Proc. of ACM Conference on Computer and Communications Security*, 2008.

[6] A. Belenko and D. Sklyarov. "secure password managers" and "military-grade encryption" on smartphones: Oh, really?, 2012.

[7] K. Bhargavan and A. Delignat-Lavaud. Web-based attacks on host-proof encrypted storage. In *Proc. of WOOT*, 2012.

[8] K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis. Language-based defenses against untrusted browser origins. In *USENIX Security Symp.*, 2013.

[9] M. Blanchou and P. Youn. Password managers: Exposing passwords everywhere, Nov 2013. `https://www.isecpartners.com/media/106983/password_managers_nov13.pdf`.

[10] J. Bonneau, C. Herley, P. C. v. Oorschot, and F. Stajano. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *Proc. of IEEE Symp. on Security and Privacy*, 2012.

[11] CNET. Editor's rating of password managers. `http://download.cnet.com/windows/password-managers/?&sort=editorsRating+asc`.

[12] O. Connelly. *WordPress 3 Ultimate Security*. Packt Publishing Ltd, 2011.

[13] D. Crockford. Adsafe. *adsafe.org*, 2011.

[14] Content security policy: W3c editor's draft, 2013. `https://dvcs.w3.org/hg/content-security-policy/raw-file/tip/csp-specification.dev.html`.

[15] P. De Ryck, M. Decat, L. Desmet, F. Piessens, and W. Joosen. Security of web mashups: a survey, 2011.

[16] P. Eckersley. How unique is your web browser? In *Privacy Enhancing Technologies*, pages 1–18. Springer, 2010.

[17] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna. Toward automated detection of logic vulnerabilities in web applications. In *USENIX Security Symposium*, 2010.

[18] A. P. Felt and D. Wagner. Phishing on mobile devices. *University of California, Berkeley*, 2011.

[19] S. Fogie, J. Grossman, R. Hansen, A. Rager, and P. D. Petkov. *XSS Attacks: Cross Site Scripting Exploits and Defense*. Syngress, 2011.

[20] Google. Content scripts, 2013.

[21] E. Grosse and M. Upadhyay. Authentication at scale. *Security Privacy, IEEE*, 11(1):15–22, Jan 2013.

[22] C. Herley. So long, and no thanks for the externalities: the rational rejection of security advice by users. In *Proc. of NSPW*, 2009.

[23] A. Huth, M. Orlando, and L. Pesante. Password security, protection, and management. *United States Computer Emergency Readiness Team*, 2012.

[24] G. Inc. Google caja—google developers. `https://developers.google.com/caja/`.

[25] B. Kaliski. PKCS #5: Password-Based Cryptography Specification Version 2.0. RFC 2898 (Informational).

[26] Lastpass. `https://lastpass.com`.

[27] LastPass. Lastpass one million user giveaway. `http://blog.lastpass.com/2011/01/lastpass-one-million-user-giveaway.html`.

[28] X. Li and Y. Xue. A survey on server-side approaches to securing web applications. *ACM Computing Surveys*, 46(4), 2014.

[29] S. Maffeis, J. Mitchell, and A. Taly. Object capabilities and isolation of untrusted web applications. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 125–140, 2010.

[30] P. Magazine". Editor's rating of password managers. `http://www.pcmag.com/products/28042?sort=er+desc`.

[31] Mozilla Developer Network. Xpconnect wrappers, 2013.

[32] Needmypassword. `http://www.needmypassword.com`.

[33] Passwordbox. `https://www.passwordbox.com`.

[34] D. Pogue. Remember all those passwords? no need. `http://nyti.ms/10ZhXgq`, 2013.

[35] Roboform everywhere. `http://www.roboform.com/everywhere`.

[36] M. Rochkind. Security, forms, and error handling. In *Expert PHP and MySQL*, pages 191–247. Springer, 2013.

[37] D. Silver, S. Jana, E. Chen, C. Jackson, and D. Boneh. Password managers: Attacks and defenses. In *Proceedings of the 23rd Usenix Security Symposium*, 2014.

[38] S. Son, K. S. McKinley, and V. Shmatikov. Rolecast: finding missing security checks when you do not know what checks are. In *ACM SIGPLAN Notices*, volume 46, pages 1069–1084. ACM, 2011.

[39] F. Sun, L. Xu, and Z. Su. Static detection of access control vulnerabilities in web applications. In *USENIX Security Symposium*, 2011.

[40] S.-T. Sun and K. Beznosov. The devil is in the (implementation) details: an empirical analysis of oauth sso systems. In *Proceedings of ACM conference on Computer and communications security*, 2012.

[41] W3C. User interface safety directives for content security policy, 2012. `http://www.w3.org/TR/UISafety/`.

[42] R. Wang, S. Chen, and X. Wang. Signing me onto your accounts through facebook and google: A traffic-guided security study of commercially deployed single-sign-on web services. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 365–379, 2012.

[43] C. Warren. Master passwords at risk in lastpass security breach. `http://mashable.com/2011/05/05/last-pass-breach/`.

[44] R. Woodbridge. "how passwordbox passed gmail as the #1 productivity app on their way to over 1m downloads". http://untether.tv/2013/episode-467, 2013.

[45] M. Zalewski. On designing uis for non-robots. `http://lcamtuf.blogspot.com/2010/08/on-designing-uis-for-non-robots.html`, 2010.

[46] W. Zeller and E. W. Felten. Cross-site request forgeries: Exploitation and prevention. Technical report, Princeton University, 2008.